

Sumario

Guillemot, con la vista en VooDoo 3

Con la vista y con algo más, pues esta compañía francesa ha anunciado ya su participación en el nuevo chipset VooDoo 3 de 3Dfx. Como ha sucedido con las anteriores versiones de este chip, Guillemot desarrollará una tarjeta aceleradora de gráficos 3D que lo incluirá. Este nuevo acuerdo hace vibrar a los "jugones" de pro, que apenas nos hemos acostumbrado todavía a las capacidades de las tarjetas VooDoo 2 cuando tenemos que imaginar un más allá. En principio, el nuevo chip ofrecerá a los usuarios las capacidades que podían conseguir dos VooDoo 2 conectadas en SLI. Además, también optimizará los gráficos 2D, como ya ha sucedido con la hermana menor Banshee. Se trata, así pues, de una noticia grata para nuestro medio, aunque tengamos que esperar mucho para que aparezcan juegos que realmente sean capaces de aprovechar tanta potencia.

Nintendo presenta una cámara para Game Boy

Los responsables de la conocida Game Boy siguen sacando productos al mercado que giran entorno a ella, con el objeto de colmar de atenciones a los muchos aficionados a esta sencilla consola portátil. Y, de paso, para poder seguir vendiendo unidades de sus muchos productos. En esta ocasión se trata nada más y nada menos que de una cámara con la que podrás realizar fotos de ti mismo en los momentos más comprometidos de tus experiencias con una consola que levanta pasiones en el mundo entero. Es decir, cuando seas derrotado en el juego más complicado de tu Game Boy. Se trata de un cartucho sobre el que se ha montado una cámara. Dicho cartucho se puede insertar en cualquier Game Boy, de modo que se convierte en una momentánea cámara digital. Para rizar el rizo, la compañía Nintendo pone a disposición de sus usuarios un complemento de su especial cámara. Se trata de una diminuta impresora que te permite imprimir las fotos que hayas hecho en un papel adhesivo. Si estos productos tienen la repercusión que se espera, tendremos que empezar a soportar a todos nuestros hijos, hermanos o vecinos intercambiándose imágenes y haciendo fotos a toda la familia.





Nintendo 64: The Legend of Zelda

Nintendo ha presentado estas Navidades un programa que se anuncia como uno de los mejores de esta consola. Lo cierto es que se trata de un gran título, como ya hemos tenido ocasión de comprobar más de uno. Es el último eslabón de una larga serie de aventuras, todas aparecidas bajo el mismo subtítulo.



El autor no es otro que Shigeru Miyamoto, un reputado cerebro creador de ojos sesgados. Este veterano maestro de los videojuegos tiene en su haber, aparte de la serie de Zelda, títulos como Super Mario 64 o Lylat Wars. El nuevo episodio de Zelda, con el subtítulo Ocarina of Time, introduce a los usuarios en un mundo tridimensional en el que el aguerrido Link tendrá que mostrar todas sus grandesdotes como aventurero.



Destacamos

Dentro de nuestro CD-Rom de portada incluimos en esta ocasión el siguiente material relacionado con la sección Game Developer:

- Los códigos fuente de los ejemplos comentados en el artículo 3D Manía.
- Los códigos fuente de los ejemplos comentados en el artículo DirectX.

Autor: Jesús de Santos
icg_ent@bigfoot.com

Bump Mapping III: Simulación Tridimensional (Bump 3d)

Con el artículo de este mes podemos dar por finalizada nuestra serie de artículos sobre la técnica Bump Mapping. Nos despedimos mostrando una aplicación de Bump Mapping en un engine 3D.

Como prometimos en nuestras anteriores entregas, hoy mostramos cómo mejorar un engine y añadirle la capacidad de mostrar superficies rugosas. Concretamente vamos a tomar nuestro engine (que hemos desarrollado totalmente desde 0 desde el comienzo de esta sección). Recordemos que el último avance que hicimos con el mismo fue el soporte de texture-Mapping y de enviroment-Mapping para simular Phong (para aquellos que quieran consultar dicho artículo, apareció en el número 12).

Ante el problema de mostrar superficies rugosas ya vimos cómo la solución de aumentar la densidad poligonal de los modelos no era un camino eficiente, al menos de momento (con la potencia actual de las máquinas). Ahora que vamos a trabajar totalmente con modelos tridimensionales podemos replantear el problema que nos surgió (al cual ya dimos solución para el caso particular de las dos dimensiones) así como dar una solución óptima.

- Problema: Con los métodos de iluminación explicados (Gouraud, Phong) y aplicando texturas que simulan relieve por ejemplo una textura de una pared de ladrillos no conseguimos un resultado realista. La luz se aplica sobre la textura como si ésta fuera plana cuando realmente no lo es, o al menos eso es lo que representa su dibujo.
- Solución ideal: Antes de aplicar la iluminación a cada píxel de la textura, modificamos ligeramente su normal; es decir, la perturbamos según un mapa de perturbaciones. Esto es sólo válido para Phong donde interpolamos la normal a lo largo de la superficie. Con esto obtenemos una iluminación mucho más realista pues la luz recorre la superficie como si ésta tuviera relieve. Por supuesto, la perturbación de las normales debe hacerse según el relieve que representa la textura a aplicar.

Otro método muy similar, es modificar directamente el punto de la superficie en vez de su normal. Así por ejemplo, sea «P» un punto de la superficie, lo modificamos sumándole su normal normalizada por un factor de perturbación (por ejemplo):

$$P' = P + BN$$

En cualquiera de los dos casos, es claro que no podemos aplicar en tiempo real estos algoritmos. Simplemente la interpolación Phong de primitivas (como ya describimos en esta sección) es demasiado costosa para los procesadores actuales e incluso para las tarjetas 3D de última hornada. Si ya es costoso el Phong, mucho más lo es la aplicación luego de la perturbación. Necesitamos un camino

FIGURA 1. RESULTADO OBTENIDO CON EL ENGINE QUE CONSTRUIMOS ESTE MES. SOLO EJECUTANDO EL VISOR APRECIAREMOS REALMENTE LA CALIDAD QUE SE CONSIGUE.



FIGURA 2. MISMA IMAGEN QUE EN LA FIGURA 1, PERO AHORA SIMPLEMENTE USAMOS ENVMAPPING. SIN EL EMPLEO DE BUMP.



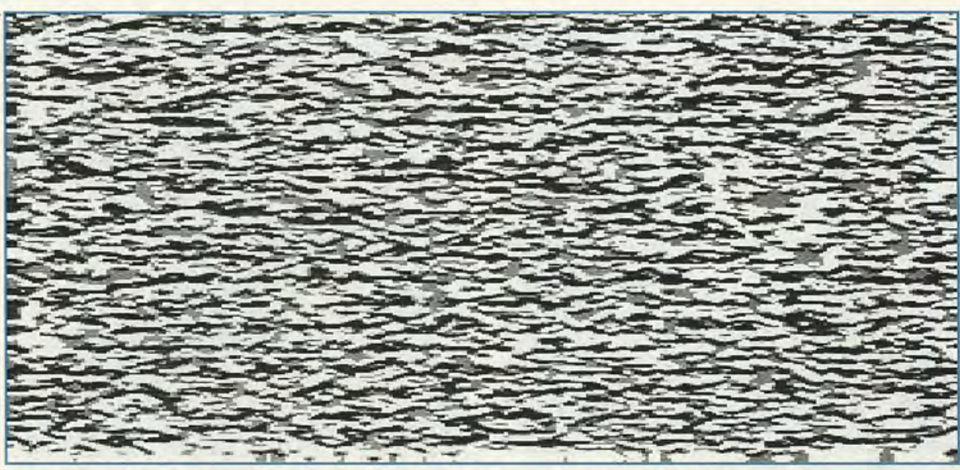


FIGURA 3. MAPA DE ALTURAS EMPLEADO PARA LA RUGOSIDAD DEL OBJETO DE LA FIGURA 1.

alternativo, que aunque sea mucho menos preciso, logre engañar al ojo humano consiguiendo un rendimiento aceptable.
Si habéis seguido esta sección estos últimos meses (y si no, es muy recomendable que lo hagáis ahora mismo) tendréis la solución...

- Solución efectiva:
  Una solución increiblemente rápida es la que vimos el mes pasado. Está intimamente relacionada con el EnvMapping, y de ahí su nombre: BumpMapping. El EnvMapping y el BumpMapping funcionan juntos muy eficientemente (tanto que las últimas tarjetas - 3D, Voodo II, TNT Riva, etc. - incorporan esta técnica en hardware [ver recuadro]). Mediante el EnvMapping nos libramos de la interpolación de las normales en cada píxel. En vez de interpolar normales simplemente simulamos la interpolación mediante una textura (todo se vio con detalle en su momento). Y es aquí donde viene el BumpMapping, que no es más que otra textura (no exactamente, como veremos un poco más adelante) que desplaza ligeramente las coordenadas de textura (u,v) del EnvMapping en cada píxel. Es justamente lo que hicimos en nuestro ejemplo 2D.

De hecho, que nadie se deje engañar por el título de este artículo, aunque nos refiramos a Bump Mapping 3D, realmente sigue siendo 2D, pues no es más que un truco sobre una textura. El título real de este artículo debería ser: BumpMapping sobre Polígonos 3D, o algo así, pero eso queda un poco feo ¿no?
Con todo lo dicho, ya podemos ponernos manos a la obra para añadir Bump3D a nuestro motor 3D. Vayamos, como siempre, paso a paso mostrando todas las modificaciones que se hagan al motor.

NUEVO MOTOR

Para simular el Bump-Mapping vamos a emplear texturas de 8 bits (y, por razones de eficiencia, de 256x256), donde cada píxel representa la altura del punto correspondiente en la textura. Pero realmente no es esto lo que necesitamos. Nosotros estamos interesados en las derivadas horizontales y verticales (dx,dy) de cada punto de altura. Para ello, tenemos que construir un procedimiento que pase de un mapa de alturas a un BumpMapping.
Destacar que si queremos emplear mapas de alturas dinámicos (es decir que varíen durante la simulación, como hicimos el mes pasado con el simulador de agua) entonces no podemos hacer este precálculo. Es necesario hacerlo en cada frame.
En el listado 1 aparece el procedimiento que hemos empleado en el ejemplo de este mes. Hemos decidido emplear un formato de BumpMapping de 16 bits, donde los 8 bits superiores son para Dy, y los 8 bits inferiores son para Dx. Ambos representados con bytes con signo (es importante no olvidar el signo).
La carga de la textura, así como su conversión a BumpMapping es trivial si empleamos nuestras rutinas de Tgas. Una vez creado el BumpMapping, ya no necesitamos la textura y podemos liberarla de la memoria principal.

```
read_tga_map("Bump.Tga",height, NULL,
Bhead.width, Bhead.height);
build_Bump(height, Bump);
free(height);
```

El siguiente problema que resolver está relacionado con la interpolación de las texturas. Necesitamos dobles coordenadas de textura para vértice:
(u,v): Coordenadas de textura 'normales'. Leídas directamente de nuestro formato G3d que importa de 3ds. Estas serán las coordenadas que vamos a emplear para acceder al BumpMapping
(u2,v2): Coordenadas de EnvMapping. Se calculan en tiempo real en función de la normal de cada uno de los vértices que tenemos y la posición de la cámara (como ya hemos visto en algún ejemplo anterior).

LISTADO 1

```
// A partir de un mapa de alturas de
dimensiones 256 x 256 x 8 bpp , //
generamos un fichero que guarda el par
(dx,dy) para cada uno de // los puntos del
mapa de alturas.

void build_Bump(BYTE *height, WORD *Bump)
{

   DWORD i,j;
   DWORD left,right,up,down;
   SBYTE dx,dy;
   WORD b;

   for(i=0;i<256;i++) for(j=0;j<256;j++) {

      if(i==0) left=0; else left=i-1;
      if(i==255) right=255; else right=i+1;

      if(j==0) up=0; else up=j-1;
      if(j==255) down=255; else down=j+1;

      dx= *(height + j*256 + right) - *(height
+ j*256 + left);
      dy= *(height + up*256 + i) - *(height +
down*256 + i);

// Empaquetar en un WORD dx y dy

      b = (dy<<8) | dx;

      *(Bump + j*256 + i) = b;

   }

   }
```

Tener dobles coordenadas de textura implica que necesitamos modificar nuestra rutina de clipping 3D para que recorte correctamente los nuevos parámetros del vértice:

```
temp[cur_temp].u2=pold[k]->u2+
(pold[j]->u2 - pold[k]->u2)*slope;

temp[cur_temp].v2=pold[k]->v2+
(pold[j]->v2 - pold[k]->v2)*slope;
```

Con todo esto ya nos encontramos perfectamente listos para implementar una rutina que, a partir de los vértices y de dos texturas, pinte en pantalla un polígono con BumpMapping:
0

```
void triangle_Bump(VRT *vrt, DWORD no_vrt,
BYTE *vid,
GFXMODE_EX *gfx,WORD *text, WORD
*Bump)
```

No tenemos espacio suficiente aquí para mostrar toda la rutina. Realmente tampoco es necesario ya que, menos el bucle de pintado, todo lo demás es una aplicación directa de temas ya tratados (interpolación de parámetros a lo largo de la superficie de un polígono en pantalla con subpíxel).
Lo realmente interesante aparece en el bucle principal, que en pseudocódigo es el siguiente:

1. Interpolar coordenadas de textura del Bump Mapping
2. Interpolar coordenadas de textura del Enviroment Mapping
3. Desempaquetar 'dx' y 'dy' desde el Bump Mapping.
4. Modificar coordenadas de textura del Enviroment Mapping :

   u <= u + dx
   v <= v + dy

5. Obtener píxel del Enviroment Mapping (con coordenadas u,v perturbadas) y pintar en pantalla.
6. Avanzar coordenadas de textura (u,v,u2,v2)
7. Volver a 3.

El código que hemos empleado para implementar este algoritmo aparece en el listado 2. Por supuesto, como siempre indicamos, para una implementación realmente eficiente sería necesario programar en ensamblador al menos esta parte.
Una vez hemos considerado todas las modificaciones, estamos en condiciones de observar los resultados finales. Para ello en la

LISTADO 2

```
for(i=0;i<(DWORD)width;i++) {

WORD b=*((WORD *)Bump +
(((cv >> 16) & 255) << 8) +
((cu >> 16) & 255));

SBYTE dx=((SBYTE)(b&255));
SBYTE dy=((SBYTE)(b>>8));

*(scan+x1+i)=*((WORD *)text
+ 8) +
(((cu2 >> 16) + dx) & 255));

cu += fdudx;
cv += fdvdx;

cu2 += fdu2dx;
cv2 += fdv2dx;

}
```


Figura 4. Mapa de entorno empleado para visualizar la Figura 1 y la Figura 2.

Figura 1 y la Figura 2 mostramos respectivamente los resultados del *engine* 3D con y sin Bump-Mapping. Hemos empleado como prueba un objeto en formato G3D con el *engine* que os mostramos hace ya tres artículos, así como en el artículo que os hemos dedicado este mes.
Brevemente vamos a comentar a continuación otra técnica que también nos permite simular Bump-Mapping con una calidad más que suficiente para lograr los objetivos que nos hemos marcado. Se trata de la técnica que, en principio, iba a ser incorporada por el programa *Quake Arena*, pero parece ser que finalmente ha sido eliminada de los planes de John Carmack. Básicamente consiste en emplear texturas a modo de mapas de luz que tienen dibujos de los relieves. Mediante una alpha-variable y en función de la posición que tenga la luz, mostramos más o menos los bordes de la textura original de la que hemos partido. Se trata de un sistema que es muy sencillo pero que nos permite obtener excelentes resultados dentro de la programación de videojuegos.


Figura 5. En la tercera parte de Quake se estuvo a punto de utilizar técnicas de Bump Mapping.

El principal incoveniente de esta técnica es que requiere numerosos mapas de luces, y además hay que realizar ciertos precálculos algo engorrosos para un programador perezoso.
Pronto, dentro de esta sección, dedicaremos unas páginas a las técnicas de mapas de luz.
Como siempre, junto al CD-Rom que acompaña a la revista incluimos el correspondiente ejemplo que hemos comentado en estas líneas, así como todas las fuentes correspondientes.
Hemos empleado el conversor de 3ds a G3d (el formato que hemos creado específicamente para esta sección y que ya describimos a fondo hace unos artículos) utilizado para convertir el modelo de tetera usado en el ejemplo.
Una vez más, insistimos en que el rumbo de esta sección depende en todo momento de vosotros, los lectores, de las sucesivas sugerencias que nos hagáis. Si tenéis alguna idea, opinión, crítica o simplemente queréis charlar con el autor de estas breves pero intensas líneas, no dudéis en contactar vía e-mail conmigo. Nos vemos en nuestra siguiente e ineludible cita.

Aprovechando el Bump 3D

La técnica que explicamos en este artículo es la más empleada para simular rugosidad en texturas: posee un equilibrio perfecto entre calidad y velocidad. Y prueba de ello es que empiezan a aparecer tarjetas aceleradoras en el mercado que soportan por hardware Bump Mapping. Así por ejemplo los chips Voodo II, y TNT Riva soportan esta innovadora característica. Para aprovechar está capacidad actualmente tenemos dos opciones.

- **Programar drivers concretos de las tarjetas (Glide por ejemplo para Voodo).**
- **Programar drivers genéricos como DirectX (la última versión es la 6.00) u OpenGl (aunque de momento, el estándar actual no soporta dicha característica).**

Así, y aunque se sale totalmente de los objetivos que, en principio, nos hemos señalado para la presente sección, bajo DirectX contamos con los siguientes identificadores de etapa de textura:

D3DTSS_BUMPENVMAT00
D3DTSS_BUMPENVLSCALE
D3DTSS_BUMPENVLOFFSET

Para más información, es recomendable recurrir al SDK que Microsoft distribuye, o a la propia dirección de e- mail del autor de estas líneas, que responde a cualquier tipo de dudas que surjan en esta sección.

Autor: Antonio Marchal

Resolución de tipos de juegos

Otra vez, como cada mes, nos encontramos en estas líneas. Como hemos ido haciendo últimamente, este mes toca otra especie de "ficha" sobre un tipo de juego. Los "comecocos", han sido famosos desde los inicios de los juegos por ordenador. Pero aunque ha tenido numerosas secuelas, no sólo vamos a hablar de *Pac-man*, como ya veremos.

Para muchos, el estridente sonido del "guaka-guaka", que traía consigo el comecocos, les sonará como la sinfonía de su músico preferido. Y es que *pacman* ha hecho historia, trayendo consigo numerosas secuelas. Algunas de ellas han diferido muchísimo de la idea original. En este articulo, únicamente se verá desde este punto de vista; es decir, teniendo como guión, el *pacman* original, donde se debían comer cocos dentro de un laberinto.

Pero los juegos que han utilizado laberintos en su base son muchos, sustituyendo los cocos por cualquier otro elemento de recolección. Algunos de vosotros llevará el tiempo suficiente en el mundo de la informática para conocer un juego llamado *Fred*, donde se generaban laberintos automáticos. Este podría entrar dentro de este grupo, dada su vertiente laberíntica. Aunque, puestos a recordar, también podríamos incluir el juego *Maziacs*, donde debías recoger un tesoro y donde se veía más representado el tipo de juegos del que vamos a hablar en el artículo de hoy.

Antes de empezar, comentar el hecho de que tomaremos el comecocos como juego base. Luego haremos un estudio de todos aquellos que tengan cierto parecido en alguna de sus partes. Y en alguno de los casos, llegaremos a un estudio más concienzudo de alguno de estos "otros" juegos. Pero dejémonos de preámbulos y entremos en materia.

JUEGOS DEL TIPO COMECOCOS Y JUEGOS ESPECIALES. FUNDAMENTOS

Antes de nada, para aquellos lectores que no sepan cuál es el juego del comecocos, daremos una pequeña descripción. El juego trataba de un protagonista redondo, con una inmensa boca que abría y cerraba cuando se movía. Con esta boca se iba comiendo unos puntos amarillos que en realidad eran cocos. De ahí el nombre de *come-cocos*, aunque originalmente era *Pac-man*, esta denominación anglosajona se podría traducir a algo así como "hombre empaquetado" u "hombre comprimido". La misión del juego era comerse todos estos cocos, para así pasar de nivel. También existían unos enemigos, con forma de fantasmas, que se debían evitar, aunque no siempre. Esto era así, porque existían unos cocos más grandes de lo normal que, cuando eran digeridos, convertían al protagonista en una especie de "Rambo". Durante un tiempo determinado, después de la ingestión de los supercocos, se permitía comerse también a los fantasmas. Durante estos periodos, los enemigos huían del protagonista en vez de perseguirlo. También existían unas frutas, que cambiaban en cada fase, que el protagonista debía intentar ingerir y que le proporcionaban puntos extras.

Una vez descritos todos los fundamentos del mítico comecocos, veremos sus aspectos uno a uno y las modificaciones que se pueden hacer. El primero de ellos es el comecocos y las paredes, es decir, los distintos laberintos que nos encontrábamos. En el caso del comemcocos, únicamente existían paredes simples, que impedían que los fantasmas cogieran al protagonista fácilmente, y viceversa. Es decir, hacían de obstáculos; pero se podría usar otro tipo de obstáculos, como son, por ejemplo, las puertas.

FIGURA 1.



Además, el nivel de complicación que existía en los laberintos del juego *Pacman*, era mínimo. Existen otros juegos en el que la complicación de éstos es por sí misma el fundamento y esencia del juego. Es más, algunos juegos se basan únicamente en laberintos, siendo su resolución la misión del juego. Se podría ir más allá, realizando un juego, como algunos que ya existen, que generara laberintos automáticamente. Es decir, que cada vez que se jugara, existiera un nuevo laberinto, como ocurría en un mítico juego español llamado *Fred*.
Pero veamos otros aspectos del juego. A continuación hablaremos de los cocos, que como hemos dicho eran los puntos que el protagonista debía comer. Se puede sustituir estos cocos por otros objetos con cualquier otra forma. Una de las más usadas es la de los tesoros, aunque también pueden ser llaves, monedas o cualquier otro objeto. Incluso, como comentábamos anteriormente, se puede prescindir de ellos.
Siguiendo por este viaje de los elementos del comecocos, llegamos a los fantasmas; es decir, los enemigos. En otros juegos se han utilizado otro tipo de enemigos. Incluso se podría llevar la idea a extremos: que dichos enemigos se comportaran como lo hacen en cualquier juego arcade e ir incluso más allá. Por ejemplo, se podrían incluir elementos móviles del juego que en determinados momentos se convirtieran en amigos y benefactores.
Esto último viene al pelo para hablar de los estados del comecocos. En el juego original, únicamente disponían de dos estados. Estaba el estado "normal" y luego el "comilon", donde podía aniquilar a los fantasmas. En otros juegos, se podrían crear más estados para el comecocos, como se ha hecho en numerosas versiones. Además, también se le podrían dar nuevas capacidades como salto, o super velocidad.

La perfecta combinación de los escasos componentes en estos juegos es la tarea más complicada

El caso de los fantasmas y enemigos es muy parecido al de los comecocos, ya que también tienen estados. Se le podrían añadir nuevos estados, a la vez de incluir nuevos enemigos, como ya se ha comentado. Con todos estos factores, el número de nuevos videojuegos que se podrían crear, combinándolos, es infinito. Por eso nos ceñiremos mayormente a la idea original, lo que no quita de que comentemos otros "nuevos" aspectos, que se pudieran implementar. Pero dejémonos de preámbulos y pasemos a ver la problemática que encierra la programación de este tipo de juegos.

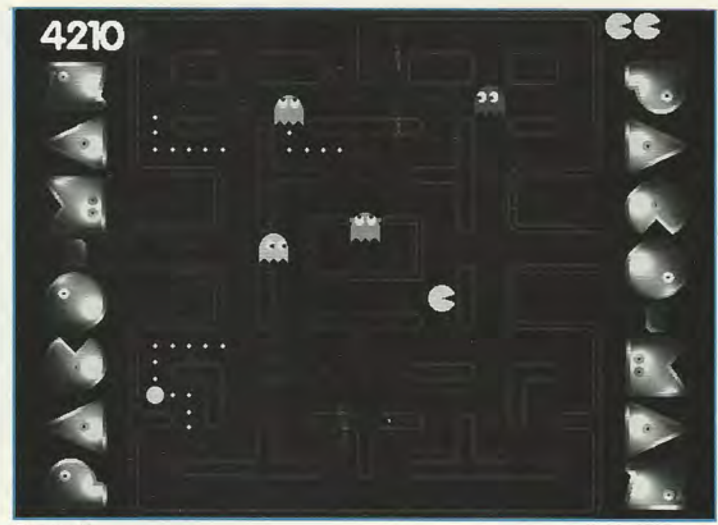


FIGURA 2.

PROBLEMATICA

Una vez vistos todos los aspectos que forman parte de los juegos del tipo comecocos y todas sus posibles vertientes, veamos todos los problemas que nos podemos encontrar. El primero de ellos son las paredes, ya que éstas forman una parte fundamental, pudiendo a llegar a casos extremos, como los generadores de laberintos. Además, se pueden usar otro tipo de obstáculos, como pudieran ser las puertas. En cualquier caso se debe tener un control sobre las paredes y todos los elementos del juego que sean parecidos. Por ejemplo, en el *Pacman*, existían unos rayos que eran una especie de puerta que impedía pasar al fantasma. En versiones posteriores se pusieron puertas o losetas con características especiales.
Y algo que tiene que ver con las losetas son los cocos, los cuales también se deberán tener controlados. La cosa se complica si se ponen otro tipo de cocos, como podrían ser tesoros, llaves o, como en el caso del comecocos, los super-cocos. Como la misión del juego es comer todos los cocos, se debe saber en todo momento cuántos cocos quedan en pantallas. Además de tener que controlar al *pacman*, leyendo el teclado y haciendo que se comporte de forma correcta cuando se encuentre cualquier obstaculo, también se debe controlar a los fantasmas, que puede parecer que tienen una programación más difícil. Se pueden usar varios métodos para hacer las rutinas de movimiento de los enemigos, además también se puede optar por poner otro tipo de enemigos. Si se dejara dar rienda suelta a la imagiación, se podrían plantear el incluir enemigos que rocen el arcade o incluso que vayan mas allá.
Una vez controlado los cocos, las paredes, los movimientos del protagonista y los enemigos, se debe pasar a los estados del comecocos. En el juego original, únicamente tenían dos estados, pero se pueden incluir muchas otras acciones, que son más comunes en otro tipo de personajes, pudiendo llegar a limites como los del protagonista del juego *Kid Kamaleon*, que tiene decenas de estados. En algunos juegos además se le han dado nuevas capacidades al protagonista, como pueden ser el salto o la super velocidad.
Por último, hay que tener en cuenta que los enemigos también tienen estados. Como se ha dicho anteriormente, se puede variar la idea original del comecocos e

El mundo de los videojuegos

Se ha abierto una nueva etapa dentro de estos artículos, con los que os entretenemos cada mes. El denominador común de todos ellos es que tratan videojuegos, comentando sus fundamentos, viendo su problemática y dando la solución a dichos problemas. Para tener todo el tema organizado se ha hecho una agrupación en tipos de videojuegos. La lista completa de esta serie de artículos es la que viene a continuación, aunque siempre puede que haya cambios de última hora, ahí va dicha lista:

- **Arcade de plataformas.**
- **Shoot'em up.**
- **Puzzles y tetris.**
- **Comecocos y juegos especiales.**
- **Juegos de cartas y de mesa.**
- **Aventuras conversacionales.**
- **Rol / RPG.**
- **Juegos de lucha.**
- **Simuladores deportivos.**
- **Simuladores de vuelo.**
- **Simuladores en general.**
- **Estrategia.**
- **Juegos tipo DOOM.**

incluir en nuestro juego nuevos estados para los enemigos. Pero pasemos a ver la solución, a la hora de ponerse a programar, que se ha buscado con todos los problemas que hemos comentado.

RESOLUCION

El primer problema es el movimiento del comecocos y, sobre todo, que detecte las paredes. Esto se puede conseguir con mapas de durezas. En el juego de ejemplo que viene con DIV, se hace de una forma parecida. Pero esta vez, se ha optado, por marcar los caminos, en vez de las paredes. Es decir, se han pintado una especie de guías, que indican las direcciones posibles en los cruces de caminos. Con esto se consiguen una serie de ventajas, a la vez que tienen otra serie de inconvenientes. Esto es así, porque se limita la capacidad de movimiento de los muñecos. Se volverá más adelante sobre este tema, cuando se hable de los movimientos de los enemigos.

Es increible la cantidad de secuelas que han tomado prestadas algunos o todos los componentes del original

Pero antes de pasar a hablar de otro tema, comentaremos algo sobre la resolución de la programación de laberintos. Ya que este es un tema que en parte se sale de los típicos juegos de comecocos únicamente se darán unas pistas que sirvan de pauta a la hora de plantearse el problema. La primera de ellas es que, si se quieren generar laberintos, el número de casilla de los lados debe ser impar. Luego se sitúa una especie de cursor, que es el que ira pintando las distintas zonas de pared y suelo en la posición (1,1), suponiendo que el mapa empieza en la (0,0). Este cursor debe ir avanzando de 3 en 3 casillas, con esto se conseguirá el ir comprobando si esa zona del laberinto esta pintada o no.

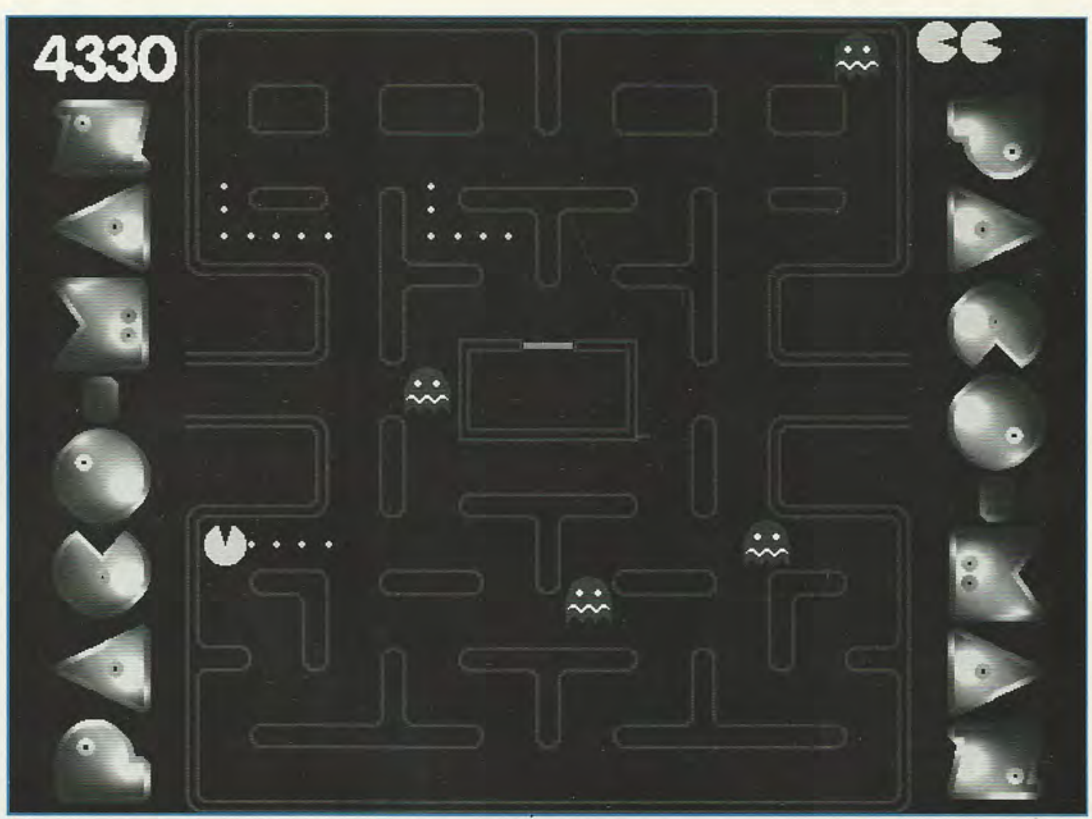


FIGURA 3.

Unicamente queda programar una rutina recursiva que recorra todo el mapa, pero no de forma secuencial, sino que debe hacerlo con una organización en una especie de árbol, donde cada rama del mismo es un nuevo camino. Una vez construido el laberinto, se debe pintar, tanto a nivel gráfico, como en el mapa de durezas.

No se ha comentado nada de otro tipo de obstáculos, como podrían ser las puertas. En estos casos, la solución más fácil es la de pintar en tiempo real el estado de la puerta en el mapa de durezas. Con eso se tendrá controlado el paso por dicho lugar y solo quedaría tener en cuenta el aspecto gráfico de dicha puerta. Esto se facilita si se hace uso de una serie de banderas que indican el estado de la puerta en cada momento.

Y es que en el juego del *Pacman*, que viene como ejemplo dentro de DIV, se han utilizado los mapas de durezas para controlar otros aspectos del juego. Uno de ellos son los cocos, que también aparecen pintados dentro del mapa de durezas. Además, se tiene una variable que hace las funciones de contador de cocos. Si se quisieran incluir otro tipo de objetos comestibles, como podrían ser los tesoros, se podría hacer de forma parecida, o también usando otros métodos.

Un poco de historia...

En el caso del juego *Pacman*, aunque parezca que no existen muchas versiones, no es así, ya que existen muchos juegos que se podrían denominar seguidores de la idea original. Lo primero es comentar que el juego original está en formato de recreativa. Por esto se han hecho innumerables secuelas para multitud de ordenadores y consolas. Pero esto no son versiones en sí, ya que se ceñían, dentro de lo posible, a la idea original.

La primera versión oficial fue *Ms-pacman*, donde cambiaron de sexo al protagonista y le dieron capacidades de super velocidad. El juego en sí era el mismo, sólo cambiaba este aspecto y, cómo no, el diseño gráfico. Luego, dentro del ordenador, se hicieron algunas versiones, donde "*Super-pacman*", con multitud de "poderes", hacía de las suyas comiendo, disparando, saltando, etc. Una de las mas reseñables, fue la realizada desde el punto de vista de las 3 dimensiones, donde se pudo integrar el salto.

Luego habría que hablar de otro tipo de juegos, que tienen que ver con los comecocos. Un de ellos podría ser el *Pengo*. En este juego se controlaba a un pingüino que debía jugar tres bloques. Aunque también se podría incluir a este juego dentro de los de Puzzle. Algo parecido pasa con *Dig-dug*, pero en el caso de los arcades. En este juego, se debía perseguir a unos monstruos e hincharlos con la bomba del protagonista hasta que explotaran. También había tesoros y "cocos" que comer.

Y ya fuera de la vision típica del juego, también se podría incluir el juego *Mummy* donde se debían recoger una serie de tesoros. Como en los casos anteriores, este juego tenía partes de aventura y puzzle. Y es que, el mezclar ideas, se ha utilizado siempre, y si no mirar el juego *Q*wert*, donde se tenía que pintar una serie de bloques. Este juego tenía cierto parecido, en cuanto a su forma de funcionar, al comecocos. Pero, el hecho de pintar en vez de comer "cocos", se ha utilizado en otros juegos, como el *Amidar*, el *Pintor* y todas sus secuelas.

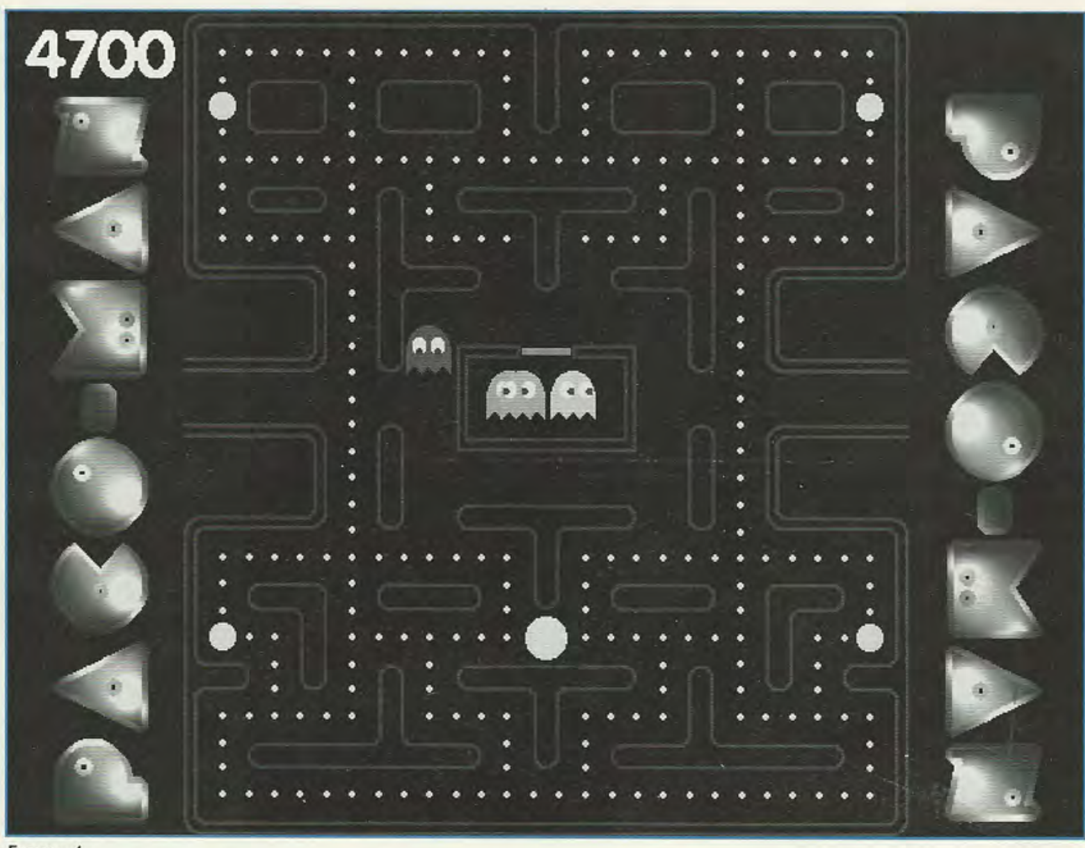


FIGURA 4.

Uno de ellos es usando procesos que manejen estos objetos. En este caso, se aprovecharían las capacidades de autonomía que dan, al utilizar variables locales, para el control de los mismos. El único problema que se puede encontrar uno es que el número de los mismos sea elevado. En estos casos es mejor pintarlos a mano, como se hace en el ejemplo de DIV.

La forma de controlar a los fantasmas, dentro de dicho ejemplo, también es bastante curiosa. Se usa también un mapa de durezas, para mover a los enemigos, indicando en cada cruce los caminos posibles que pueden tomar. Además, existe otro mapa de durezas que indica el camino hacia la casa de los fantasmas. Esto es muy útil cuando los enemigos son comidos por *Pacman*. En estos casos, únicamente se debe leer la guía pintada en el mapa de durezas, para saber la dirección correcta.

En el caso de usar otro tipo de enemigos, se debe tener en cuenta el tipo de comportamiento a usar. Si es muy parecido al que se usa para el movimiento de los fantasmas tradicionales, lo mejor es la utilización de guías en un mapa de durezas. Si no es así hay que sopesar la posibilidad de si es compatible el nuevo enemigo con el tipo de juego. Y en caso de que la integración sea posible, programar las nuevas posibilidades. Por ejemplo, imaginemos que se quiera incluir un nuevo fantasma que dispare proyectiles. Además de comportarse como los demás fantasmas, se debe programar el código que controle todo el proceso de las balas.

Los generadores de laberintos son los más interesantes creadores de paredes

Y es que, el incluir nuevos aspectos a un juego a veces no es posible y, sin embargo, otras veces es lo más sencillo del mundo. Esto pasa, por ejemplo, con los estados del comecocos, que en el juego inicial son dos. El estado normal no tiene ningún tipo de complicación, siendo algo más complicado el otro estado, el de "comer". Este caso, únicamente hace falta tener un contador de tiempo que controle la duración de dicho estado y que actualice el proceso al nuevo estado cuando sea necesario.

Si se quisiera programar otras acciones, al igual que en caso de los fantasmas, habría que sopesar si es posible o no. Hay algunas, como el salto o la super velocidad, que no entrañan mucha complicación. La última de ellas se resolvería cambiando la cantidad de incremento o, en otras palabras, controlando la velocidad con una variable, que se modificaría cuando fuese necesario. En el caso del salto, hay que tener en cuenta también que debe quedar bien en el aspecto gráfico. Por eso, en algunos juegos que han incluido esta posibilidad, se ha optado por una visión en 3D, para que así se pueda ver claramente cuando el protagonista hace un salto.

Por último habría que tener en cuenta los estados de los fantasmas. O si se diera el caso, las nuevas posibilidades que se han integrado al juego en el aspecto de los enemigos. Tanto en el caso anterior, para los estados del protagonista, como en éste, lo mejor es controlar todo mediante una variable local, a la que podríamos llamar *estado*. Dentro del código del proceso, se debería controlar y actualizar dicha variable, realizando unas acciones u otras en cada momento.

Con esto quedarían vistas todas las distintas partes de un juego del tipo comecocos. Muchos de los aspectos nuevos que se han comentado también forman parte de otro tipo de juegos. E incluso, en algunos casos, son la parte fundamental de los mismos. Por eso únicamente se ha comentado por encima, dando sólo algunas pistas sobre su resolución. Pero para el caso del comecocos, lo mejor es remitirse al ejemplo que viene con DIV Games Studios, donde se ve de forma práctica todo lo comentado en el artículo.

Resumiendo

Aunque este tipo de juegos, comparado con otros, es poco extenso, se han hecho muchísimas versiones del mismo, mezclados con una gran cantidad de variantes. Por eso nos hemos ceñido a la idea original, dando sólo pistas en la mayoría de los casos. Además, se dispone del listado completo, que viene dentro del ejemplo de DIV. Es recomendable su estudio para la gente que le interese. Saber que siempre podéis escribir a la dirección de e-mail: *tizo@100mbps.es*, donde todas vuestra dudas y sugerencias serán bienvenidas. Bueno, nos despedimos hasta el mes que viene, donde un nuevo año, y un nuevo DIV, aparecerán en nuestras vidas. Hasta entonces, espero que hayáis tenido y tengáis una buena entrada de año, y que os DIVirtáis programando.

Autor: José Antonio Guerra
balder@mindless.com

Sonido en DirectX

Tranquilos. No os asustéis. De siempre la programación del sonido ha sido algo más bien engorroso, sobre todo por la necesidad de hacer códigos para varios dispositivos distintos. Pero ahora todo ha cambiado; tenemos a nuestra entera disposición DirectSound, el componente de DirectX que nos va a facilitar enormemente estas tareas.

DirectSound es el elemento de las librerías DirectX que permite el uso de los dispositivos de sonido. Tal y como ocurría en DirectDraw, el modelo de abstracción de dispositivos nos permite olvidarnos de la pesada tarea de programar cada tarjeta de sonido por separado sin renunciar a la velocidad. Por supuesto, el uso de DirectSound nos permitirá aprovechar características avanzadas de las últimas generaciones de tarjetas de sonido sin tener que preocuparnos por el modelo o marca del dispositivo instalado en cada ordenador.

Otra de las ventajas de DirectSound es que está diseñado para ser rápido. El conjunto de rapidez y abstracción permite aprovechar al máximo tanto dispositivos de sonido modestos como los más avanzados. ¡Y todo esto de un modo transparente al desarrollador! Uno de los detalles que más llama la atención de esta arquitectura es el hecho de poder implementar funciones en nuestros programas pensando en futuros dispositivos. Imaginemos un juego 3D con control del nivel de detalle. En el detalle más alto, no hay aceleradora ni procesador que lo soporte, aunque previsiblemente sí exista en un año. Pasado el año, cuando la aceleradora estuviese disponible, el programa la puede aprovechar y mostrarse en todo su esplendor. Y todo sin necesidad de parches, ya que la gran mayoría de dispositivos salen al mercado con su controlador DirectX correspondiente.

DirectSound permite aprovechar al máximo las características de hardware de los dispositivos de sonido, ya que es posible obtener información detallada sobre las funciones soportadas. Así podremos ajustar el uso del sonido a la configuración actual de cada ordenador. Por ejemplo, en tarjetas de sonido con memoria RAM y mezcla de sonidos por hardware podremos cargar todas las muestras en dicha memoria y hacer que el dispositivo efectúe la mezcla, liberando al procesador de un importante tiempo de proceso. Asimismo, si el dispositivo no dispone de estas funcionalidades, podemos reducir el número de sonidos a mezclar simultáneamente para así permitir a la aplicación correr más eficientemente.

EDITANDO SONIDOS CON COOL EDIT.



DirectSound también permite la reproducción de sonidos 3D. Las interfaces IDirectSound3DBuffer e IDirectSound3DListener permiten gestionar y reproducir sonidos teniendo en cuenta parámetros espaciales como posición y velocidad. Además, gestiona efectos físicos reales como el efecto *Doppler*, atenuaciones y conos de audición. Estas interfaces están especialmente diseñadas para ser usadas en conjunción con Direct3D y crear ambientes sonoros reales en aplicaciones tridimensionales.

La captura de sonido está también contemplada en DirectSound. Mediante las interfaces IDirectSoundCapture e IDirectSoundCaptureBuffer podremos capturar sonido para nuestras aplicaciones. Hay que notar que estas interfaces simplemente encapsulan llamadas a las funciones multimedia de Win32, con lo cual por ahora no dependen en absoluto de los controladores DirectX. Es de esperar que en versiones posteriores de las librerías sí se implementen estas funcionalidades directamente en los controladores.

ARQUITECTURA DIRECTSOUND

DirectSound está formada por las siguientes interfaces:

- *IDirectSound*: Creación de objetos DirectSound e interacción con el entorno.
- *IDirectSound3DBuffer:* Buffers de sonido para aplicaciones 3D. Permite parámetros de velocidad, posición y conos de audición, así como atenuación y varios efectos.
- *IDirectSound3DListener:* Describe la posición y el entorno del oyente en aplicaciones con sonido 3D.
- *IDirectSoundBuffer:* Manipulación de buffers de sonido.
- *IDirectSoundCapture:* Creación de buffers de captura.
- *IDirectSoundCaptureBuffer:* Manipulación de buffers de captura.
- *IDirectSoundNotify:* Mecanismo que permite establecer eventos de notificación para buffers de sonido.
- *IKsPropertySet:* Obtención de información sobre propiedades extendidas de los dispositivos de sonido.

BUFFERS DE SONIDO

El objeto IDirectSoundBuffer representa un buffer que contiene datos de sonido en formato PCM. Mediante él podemos reproducir o modificar dicho sonido, además de poder cambiar atributos como el formato de onda (frecuencia de muestreo, número de canales), el volumen, la frecuencia de reproducción y el balance de canales (panning).

Hay dos tipos de buffers de sonido; los primarios y los secundarios. El buffer primario representa el sonido que se reproduce a través del dispositivo de sonido y es creado automáticamente al iniciar el objeto *IDirectSound*. Los buffers secundarios almacenan los sonidos que usaremos en nuestra aplicación y serán mezclados automáticamente

EL PROGRAMA DE EJEMPLO.

```
DirectSound - Curso DirectX - Entrega 7
Pulsa las siguientes teclas para probar:
[ESC] para salir


[1]  -  Buffer original
[2]  -  Buffer duplicado
```

al buffer primario cuando sean reproducidos. Estos buffers pueden almacenarse en la memoria del sistema o en la del dispositivo, si éste lo permite. En este último caso la mezcla de sonidos será realizada por el hardware, liberando tiempo de proceso. En caso contrario, la mezcla es efectuada por DirectSound automáticamente y el único límite en el número de sonidos que se pueden mezclar simultáneamente es el tiempo de proceso de la CPU.

Es importante mencionar que DirectSound no incluye funciones para cargar ficheros de sonido, por lo cual deberemos hacerlo nosotros mismos. Como veréis más adelante, en nuestro programa de ejemplo hemos usado sonidos sin formato (raw).

El objeto DirectSound puede ser creado por varias aplicaciones usando el mismo dispositivo de sonido. Cuando saltamos de una aplicación a otra, la salida de audio cambia a la que se vuelva activa en ese momento, con lo cual no hemos de preocuparnos en parar y volver a reproducir el sonido cuando eso ocurra. De eso se encarga automáticamente DirectSound.

INICIANDO DIRECTSOUND

El primer objeto que deberemos crear para dotar de sonidos a nuestras aplicaciones es el objeto *IDirectSound*. El paso para obtenerlo es haciendo una llamada a la función *DirectSoundCreate*, definida en dsound.h. Aquí vemos el prototipo de dicha función:

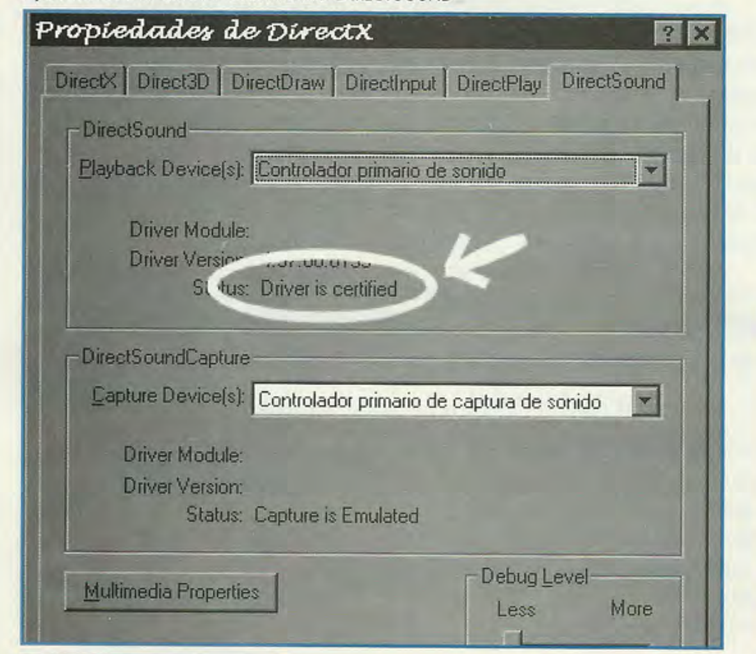
```
HRESULT DirectSoundCreate(
  LPGUID lpGuid,
  LPDIRECTSOUND * ppDS,
  IUnknown FAR * pUnkOuter
);
```

- *lpGuid:* Dirección del GUID del dispositivo sobre el que queremos crear el objeto *DirectSound*. Podemos usar uno de los valores devueltos por la función *DirectSoundEnumerate* o simplemente pasarle NULL. En éste último caso, el objeto representará al dispositivo por defecto.
- *ppDS:* Dirección del puntero en el que será devuelto el objeto *DirectSound* creado.
- *pUnkOuter:* Usado para agregaciones en objetos COM. Este valor debe ser NULL.

Veamos un sencillo ejemplo. Si queremos crear el objeto DirectSound con el dispositivo por defecto, lo cual es lo más recomendable para la mayoría de las aplicaciones, escribimos:

```
LPDIRECTSOUND lpDirectSound;
HRESULT       hr;
hr = DirectSoundCreate(NULL, &lpDirectSound, NULL));
```

Ejemplo de dispositivo certificado DirectSound



Funciones de la interfaz IDirectSoundBuffer por categoría

INFORMACION	GetCaps GetFormat GetStatus SetFormat
GESTION DE MEMORIA	Initialize Restore
CONTROL DE REPRODUCCION	GetCurrentPosition Lock Play SetCurrentPosition Stop Unlock
GESTION DE SONIDO	GetFrequency GetPan GetVolume SetFrequency SetPan SetVolume

Tras esta llamada, obtendremos el objeto DirectSound listo para funcionar en la variable lpDirectSound. Ya tenemos el primer paso, ahora veamos que debemos de hacer a continuación.

NIVELES DE COOPERATIVIDAD

Tras crear nuestro objeto *DirectSound*, deberemos establecer su nivel de cooperatividad mediante la función *IDirectSound::SetCooperativeLevel* antes de poder reproducir ningún sonido. Vamos a ver el prototipo de dicha función:

```
HRESULT SetCooperativeLevel(
  HWND hwnd,
  DWORD dwLevel
);
```

- *hwnd:* Handle de la ventana de la aplicación.
- *dwLevel:* Indica el nivel de prioridad deseado. Puede ser cualquiera de estos cuatro valores: DSSCL_EXCLUSIVE, DSSCL_NORMAL, DSSCL_PRIORITY y DSSCL_WRITEPRIMARY.

El nivel de cooperatividad indica el modo mediante el cual nuestro objeto *DirectSound* se relaciona con el sistema operativo y los dispositivos. Windows, al ser multitarea, debe saber cómo tratar las distintas aplicaciones, sobre todo por el hecho de que dos aplicaciones distintas pueden crear objetos DirectSound sobre un mismo dispositivo. El establecimiento de los niveles de cooperatividad es fundamental para que una aplicación que comparta dispositivo con otra no acceda a éste en un momento que no deba o de un modo que no le esté permitido.

DirectSound define cuatro niveles

Funciones de la interfaz IDirectSound por categoría

GESTION DE MEMORIA	Compact Initialize
CREACION DE BUFFERS	CreateSoundBuffer DuplicateSoundBuffer SetCooperativeLevel
CAPACIDADES DE DISPOSITIVO	GetCaps
CONFIGURACION DE ALTAVOCES	GetSpeakerConfig SetSpeakerConfig

de cooperatividad; *normal, exclusive, priority y write primary*. El modo más común es el normal, ya que permite el cambio de una aplicación a otra más fácilmente. Veamos cada nivel en detalle:

Nivel normal (DDSCL_NORMAL)
En este nivel de cooperatividad, la aplicación no puede cambiar el formato del buffer de sonido primario, ni escribir en él directamente, ni compactar la memoria del dispositivo de sonido. Todas las aplicaciones con este nivel establecido usan un buffer primario con frecuencia de 22Khz, sonido estéreo y 8 bits por muestra. Esto permite cambiar entre aplicaciones que usan este nivel de una manera rápida y eficaz.

Nivel priority (DDSCL_PRIORITY)
En este nivel, las aplicaciones poseen prioridad sobre recursos del dispositivo, como la mezcla por hardware. Asimismo, pueden establecer el formato del buffer primario y compactar la memoria del dispositivo.

Nivel exclusive (DDSCL_EXCLUSIVE)
En el nivel exclusivo, la aplicación posee todos los privilegios del modo *priority*. Adicionalmente, cuando la aplicación está activa, sus buffers son los únicos que se pueden escuchar.

Nivel write primary (DSSCL_WRITEPRIMARY)
Este es el nivel más alto de cooperatividad. Una vez establecido, la aplicación tiene acceso al buffer primario y debe escribir en él, ya que, mientras tanto, no se pueden utilizar los buffers secundarios.
Cuando una aplicación en este modo se vuelve activa, los buffers secundarios de las demás aplicaciones que estén usando el mismo dispositivo son parados y marcados como perdidos. Asimismo, cuando la aplicación vuelve a estar en el fondo, su buffer primario es marcado como perdido.
Un punto importante es la imposibilidad de establecer este nivel si no existe un controlador DirectSound para el dispositivo, como por ejemplo cuando es emulado. Para determinar si se da el caso, hay que llamar a la función IDirectSound::GetCaps y comprobar el flag DSCAPS_EMULDRIVER en la estructura DSCAPS.

CAPACIDADES DEL DISPOSITIVO

Una aplicación puede obtener información detallada sobre las características de un dispositivo de sonido mediante el método *IDirectSound::GetCaps*. Aunque DirectSound aprovecha automáticamente los recursos de los dispositivos, como la mezcla de buffers por hardware, esta cualidad nos permite optimizar ciertos aspectos del sistema de sonido. Por ejemplo, una aplicación podría mezclar más sonidos simultáneamente si dispone de mezcla por hardware, ya que no consume tiempo de proceso.
Veamos un ejemplo de esta función:

```
DSCAPS dscaps;

dscaps.dwSize = sizeof(DSCAPS);
HRESULT hr = lpDirectSound->GetCaps(&dscaps);
```

La estructura *DSCAPS* es rellenada con datos acerca del dispositivo de sonido. Mediante esta información (ver tablas), obtenemos un control total de dicho dispositivo y podemos escalar los requerimientos de sonido de nuestra aplicación respecto a las características de la máquina.
Nótese que es imprescindible iniciar el parámetro *dscaps.dwSize* para que la función se ejecute correctamente.
Un fallo muy común en la programación DirectSound es crearse suposiciones sobre el comportamiento de los dispositivos. Siguiendo ese modo de trabajo, lo más fácil es que la aplicación funcione en unas máquinas y en otras no, ya que hay un gran rango de dispositivos de sonido distintos en el mercado, aparte de los futuros que irán apareciendo, cada uno de ellos con un funcionamiento distinto. El único método correcto es fijarse en las capacidades de cada uno mediante *IDirectSound::GetCaps*.

Como último, es recomendable verificar los recursos libres del dispositivo antes de la creación de cada buffer. Con estas pequeñas directrices nos aseguramos un aprovechamiento y una seguridad óptimas en DirectSound.

CREACIÓN DE BUFFERS

Veamos ahora la función *IDirect Sound::CreateSoundBuffer*. para crear *buffers* de sonido secundarios.

Valores del parámetro DSCAPS.dwFlags

El miembro dwFlags especifica las capacidades del dispositivo. Puede ser una o más de las siguientes:

DSCAPS_CERTIFIED
Indica que el controlador ha sido probado y certificado por Microsoft.

DSCAPS_CONTINUOUSRATE
El dispositivo soporta todas las frecuencias de muestreo indicadas entre los miembros dwMinSecondarySampleRate y dwMaxSecondarySampleRate. Normalmente, esto indica que la frecuencia de salida actual tendrá un margen de error de +/- 10 hertzios (Hz) respecto de la frecuencia especificada.

DSCAPS_EMULDRIVER
El dispositivo no posee un controlador DirectSound, por tanto está siendo emulado mediante las funciones Win32 de sonido. En estos casos, es probable notar una degradación de rendimiento.

DSCAPS_PRIMARY16BIT
El dispositivo soporta buffers primarios con muestras de 16 bits.

DSCAPS_PRIMARY8BIT
El dispositivo soporta buffers primarios con muestras de 8 bits.

DSCAPS_PRIMARYMONO
El dispositivo soporta buffers primarios monofónicos.

DSCAPS_PRIMARYSTEREO
El dispositivo soporta buffers primarios estereofónicos.

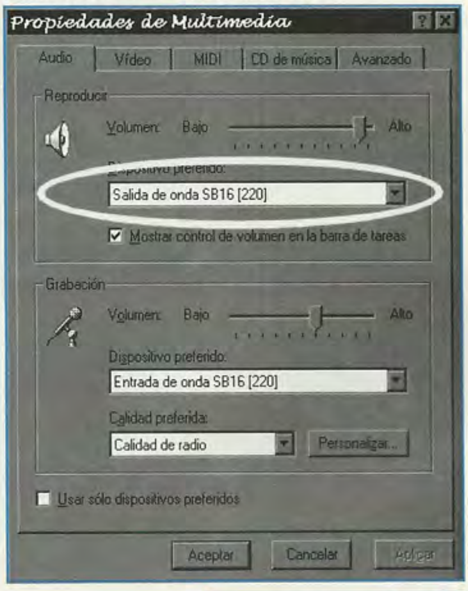
DSCAPS_SECONDARY16BIT
El dispositivo soporta buffers secundarios mezclados en hardware con sonido de 16 bits.

DSCAPS_SECONDARY8BIT
El dispositivo soporta buffers secundarios mezclados en hardware con sonido de 8 bits.

DSCAPS_SECONDARYMONO
El dispositivo soporta buffers secundarios monofónicos.

DSCAPS_SECONDARYSTEREO
El dispositivo soporta buffers secundarios estereofónicos.

DISPOSITIVO USADO EN LA EMULACION



```
HRESULT CreateSoundBuffer(
  LPCDSBUFFERDESC
lpcDSBufferDesc,
  LPLPDIRECTSOUNDBUFFER
lplpDirectSoundBuffer,
  IUnknown FAR * pUnkOuter
);
```

- *lpcDSBufferDesc:* Dirección de una estructura *DSBUFFERDESC* que contiene la descripción del buffer de sonido a crear.
- *lplpDirectSoundBuffer:* Dirección del puntero al objeto *DirectSoundBuffer* creado, o NULL si hay un error.
- *pUnkOuter:* Debe valer NULL.

Como introducción a la creación de *buffers*, tenemos un listado en el que crearemos un *buffer* de sonido de tres segundos, al que podremos modificar *panning*, volumen y frecuencia de reproducción.

```
BOOL CrearBuffer(
   LPDIRECTSOUND lpDirectSound,
   LPDIRECTSOUNDBUFFER *lplpDsb)
{
   PCMWAVEFORMAT pcmwf;
   DSBUFFERDESC dsbdesc;
   HRESULT hr;
   // Establecemos el formato de
onda
   memset(&pcmwf, 0,
sizeof(PCMWAVEFORMAT));
   pcmwf.wf.wFormatTag =
WAVE_FORMAT_PCM;
   pcmwf.wf.nChannels = 2;
   pcmwf.wf.nSamplesPerSec =
22050;
   pcmwf.wf.nBlockAlign = 4;
   pcmwf.wf.nAvgBytesPerSec =
      pcmwf.wf.nSamplesPerSec *
pcmwf.wf.nBlockAlign;
   pcmwf.wBitsPerSample = 16;
   // Establecemos la estructura
DSBUFFERDESC
   memset(&dsbdesc, 0,
sizeof(DSBUFFERDESC)); // Zero it
out.
   dsbdesc.dwSize =
sizeof(DSBUFFERDESC);
   // Necesitamos los controles por
defecto (panning, volumen,
frecuencia)
   dsbdesc.dwFlags =
DSBCAPS_CTRLDEFAULT;
   // Buffer de 3 segundos
   dsbdesc.dwBufferBytes = 3 *
pcmwf.wf.nAvgBytesPerSec;
   dsbdesc.lpwfxFormat =
(LPWAVEFORMATEX)&pcmwf;
   // Creamos el buffer
   hr = lpDirectSound->lpVtbl-
>CreateSoundBuffer(lpDirectSound,
      &dsbdesc, lplpDsb, NULL);
   if(DS_OK == hr) {
      // Sin errores. El interface válido
se almacena en *lplpDsb.
      return TRUE;
   } else {
      // Fallo
      *lplpDsb = NULL;
      return FALSE;
   }
}
```

Miembros de la estructura DSCAPS

DWSIZE
Tamaño de la estructura, en bytes. Es necesario establecer esta variable antes de usar la estructura.

DWFLAGS
Ver tabla aparte.

DWMINSECONDARYSAMPLERATE Y DWMAXSECONDARYSAMPLERATE
Frecuencias de muestreo mínima y máxima soportadas por los buffers secundarios en hardware.

DWPRIMARYBUFFERS
Número de buffers primarios soportados. Siempre vale 1.

DWMAXHWMIXINGALLBUFFERS
Especifica el número total de buffers que pueden ser mezclados en hardware. Este miembro puede ser menor que la suma de dwMaxHwMixingStaticBuffers y dwMaxHwMixingStreamingBuffers.

DWMAXHWMIXINGSTATICBUFFERS
Especifica el número máximo de buffers estáticos de sonido.

DWMAXHWMIXINGSTREAMINGBUFFERS
Especifica el número máximo de buffers de flujo de sonido.

DWFREEHWMIXINGALLBUFFERS, DWFREEHWMIXINGSTATICBUFFERS, AND DWFREEHWMIXINGSTREAMINGBUFFERS
Descripción del número de buffers libres para mezcla por hardware. Una aplicación puede usar esos valores para determinar si hay suficientes recursos para usar un buffer de sonido en hardware. También puede ser usado para determinar los recursos en uso, comparando estos valores con los que nos indican las capacidades totales del dispositivo.

DWMAXHW3DALLBUFFERS, DWMAXHW3DSTATICBUFFERS, Y DWMAXHW3DSTREAMINGBUFFERS
Descripción de las capacidades de sonido 3D por hardware.

DWFREEHW3DALLBUFFERS, DWFREEHW3DSTATICBUFFERS, Y DWFREEHW3DSTREAMINGBUFFERS
Descripción de los recursos de sonido 3D libres en hardware.

DWTOTALHWMEMBYTES
Tamaño en bytes del total de memoria RAM del dispositivo de sonido.

DWFREEHWMEMBYTES
Tamaño en bytes de la memoria RAM libre del dispositivo.

DWMAXCONTIGFREEHWMEMBYTES
Tamaño, en bytes, del bloque más grande de memoria contigua disponible en la tarjeta de sonido.

DWUNLOCKTRANSFERRATEHWBUFFERS
Descripción de la velocidad, en bytes por segundo, a la cual los datos son transferidos a buffers estáticos en la memoria del dispositivo. Este dato y el número de bytes transferidos determinan la duración de una llamada al método IdirectSoundBuffer::Unlock.

DWPLAYCPUOVERHEADSWBUFFERS
Descripción del tiempo de proceso, expresado en porcentaje respecto a la CPU, que se necesita para mezclar buffers en software (localizados en la memoria del sistema). Este valor varía dependiendo del tipo de bus, el procesador y la velocidad de reloj. Nótese que la tasa de transferencia para la función Unlock para buffers en software vale 0, ya que los datos no se transfieren a ninguna parte. Asimismo, el tiempo de proceso usado en mezclar buffers en hardware es también 0 ya que la mezcla la efectúa el dispositivo de sonido.

DWRESERVED1 Y DWRESERVED2
Reservados para uso futuro.

EL MES QUE VIENE

Tras la lectura del código, nos podemos hacer una idea del proceso de creación de *buffers* secundarios. En la siguiente entrega profundizaremos en el tema y veremos lo que podemos hacer con los *buffers* de sonido. Practicad con las capacidades de los dispositivos y con el pequeño programa de ejemplo, que podréis encontrar en el CD de la revista.
Las preguntas, sugerencias y críticas sirven para enriquecer esta sección. No dudéis en mandar vuestros mensajes al autor.



Autor: Carlos Gzlez. Morcillo

Curso de animación [II]

En nuestra segunda entrega repasaremos las principales técnicas de animación, algunos conceptos básicos y empezaremos a trabajar con un ejemplo práctico: crearemos un personaje desde cero y lo colorearemos con Photoshop.

Antes de meternos de lleno en términos nuevos, haremos un repaso obligado a los conceptos básicos de animación que ya hemos visto. A los lectores habituales de esta revista les remitimos al Game Over número 6 y 7, en la sección Taller 2D. Para los que se incorporan "nuevos" repasaremos brevemente algunos puntos clave que necesitaremos recordar.

RECORDANDO, QUE ES GERUNDIO

El animar un personaje no significa simplemente "moverlo por la pantalla". Para animarlo realmente tenemos que dotarlo de vida. Ese conjunto de *pixel* de distintos colores que formarán un muñequito en pantalla deberán simular que tienen vida propia. Para ello, definiremos previamente las características del personaje, sus fobias, su manera de andar, sus gestos... cuando tengamos todo esto, comenzaremos a animar recordando los 10 principios básicos de la animación. En este pequeño resumen citaremos los que nos hacen falta para animar personajes por separado (y no secuencias complejas), remitiendo a los lectores al número 6 de Game Over para recordarlos todos más extensamente:

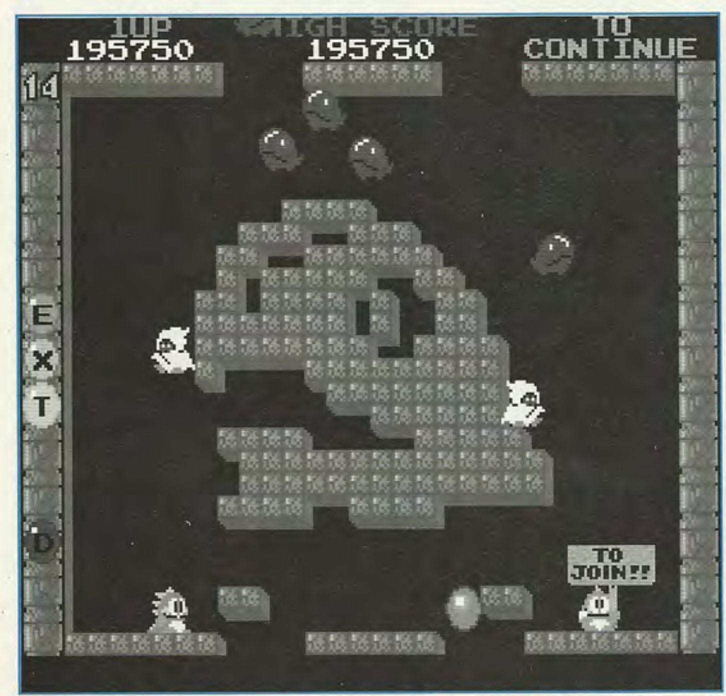
Aplastar y estirar: Nuestros personajes tienen masa y flexibilidad. Por estas características sufrirán deformaciones en su movimiento. Recordemos el ejemplo de la pelota de goma botando; cuando choque con el suelo se aplastará y saldrá rebotada estirándose (su volumen será el mismo en toda la animación). Si simplemente la movemos sin aplicarle este principio, no parecerá que esté botando.

Temporización: Mediremos la rapidez con que ocurre la acción. Este punto será crítico para diferenciar personajes que, con igual apariencia, tengan distintas características que no se ven (como por ejemplo la masa). Además, en una acción distinguiremos partes con diferentes curvas de velocidad (como un personaje que está corriendo; cuando comienza a correr y cuando para, la acción es mucho más lenta que el fugaz paso por la zona intermedia de la carrera). Este principio lo veremos con más detalle el mes que viene.

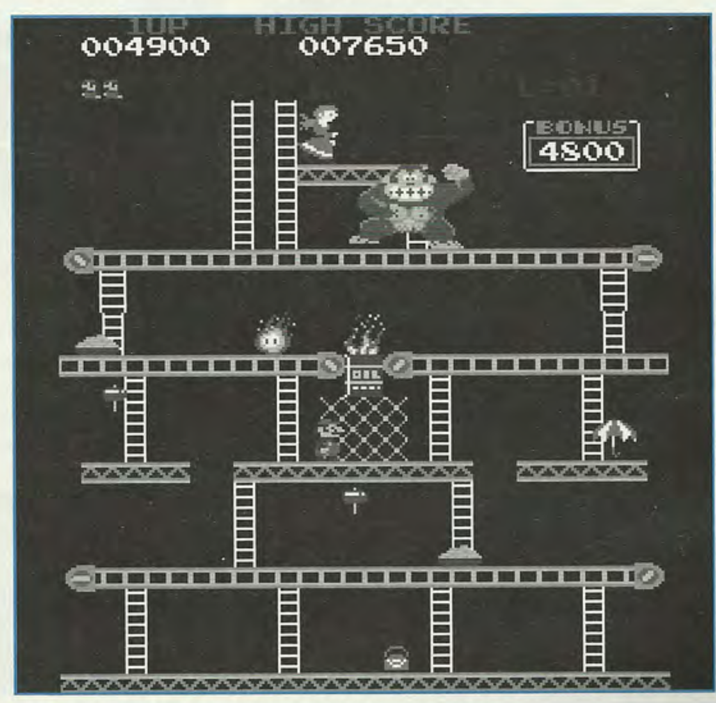
Exageración: Tanto en animaciones cómicas como para las más realistas, la exageración de movimientos (tanto de cuerpo entero como los faciales) es fundamental para facilitar al animador el dibujar ese movimiento y representar más fácilmente los sentimientos del personaje.

Arcos: Muy pocos movimientos en la naturaleza transcurren en línea recta. Por lo general, casi todos ocurren en parábolas y debemos esforzarnos por no seguir trayectorias rectas (que sería lo más inmediato al animar pero que no se corresponde con la realidad).

Atractivo y naturalidad: El animador no debe caer en fallos comunes como hacer movimientos totalmente simétricos, que los brazos y las piernas de nuestro personaje estén en la misma posición, olvidar el balanceo del cuerpo cuando anda...



El mitico Bubble Bobble, de la época de los ordenadores de 8 bits.



Un clasico que todavia sobrevive en nuestro recuerdo.





LAS DISTINTAS FASES EN EL DISEÑO DE NUESTRO LEONCILLO.

Recordemos que un *frame* (o cuadro) es cada uno de los dibujos individuales que forman parte de una animación. Un key*frame* clave (kfc) es un punto crítico de una animación. En el bote de una pelota contra el suelo, un kfc sería el punto más alto de su trayectoria antes de caer, otro justo cuando choca contra el suelo y el último cuando de nuevo ha ascendido y está en el punto más alto de su trayectoria. El espacio que hay entre dos kfc (el número de *frames* que hay entre ellos) no tiene porqué ser constante. De hecho, si pusiéramos más *frames* cerca de los kfc que en el medio, conseguiremos un movimiento fugaz en el centro y más lento en los extremos.
En los estudios de animación profesionales, un artista principal dibujará sólo los kfc, y un equipo de dibujantes "sudarán tinta" para dibujar los *frames* intermedios en cada acción. A nosotros nos tocará realizar todo el proceso (a menos que podamos contratar a alguien que haga el "trabajo sucio")

CÓMO TRABAJAREMOS

La técnica que utilizaremos para animar es la usada por la animación tradicional. Es la denominada técnica de capas (o cáscaras de cebollas). Tradicionalmente se han utilizado acetatos para que el animador dibujara en cada placa de acetato un cuadro de la animación. Esto tiene la ventaja de poder ver en cada momento los cuadros anteriores de la animación. Muchos programas de animación en 2D permiten simular esta técnica (como Autodesk Animator Pro y Macromedia Director). Sin embargo, nosotros optaremos por dibujar cada cuadro en papel sobre todo porque es más cómodo dibujar de esta forma y no directamente en la pantalla (aún cuando tenemos una tabla digitalizadora).
Para ahorrarnos trabajo, en algunos movimientos utilizaremos la técnica de los cortes externos (cut-outs). Cuando el movimiento de un personaje es limitado (como cuando habla), tan sólo redibujaremos la parte del personaje que cambia (en este caso la boca). Esta técnica es muy utilizada por los reyes de la reutilización: los animadores japoneses en sus vídeos anime-manga. En estas películas suelen reutilizar escenas hasta la saciedad, y sólo animan partes nuevas cuando es realmente necesario.

Cuando escaneemos un dibujo para trabajar en el ordenador, lo haremos aproximadamente 2 ó 3 veces más grande de como vaya a quedar al final. Será más



CERRAMOS SUPERFICIES CON EL MISMO COLOR QUE SE RELLENARAN.

cómodo trabajar con un dibujo grande, colorearlo y reducirlo al tamaño que necesitemos una vez que esté acabado. Para el dibujo del ejemplo, que veremos más adelante, se escaneó a 300 dpi (puntos por pulgada) y después se redujo a 450x800. Los defectos que tenía al colorearse y por el escaneado no se notaban nada al reducirlo. Cuando los personajes sean para un videojuego (rondarán por los 100x200 *pixel* como mucho) tendremos que retocarlos pixel a pixel antes de darlos por concluido. Antiguamente, en juegos como *Donkey Kong, Bubble Bobble*... se dibujaban todos los gráficos pixel a pixel, porque la complicación en los mismos tampoco podía ser muy grande (un ZX Spectrum con sus 256x192 o un Comodore 64 con 320x200 no daban para mucho). Actualmente, y con las herramientas de edición que disponemos, sería un trabajo de chinos (y poco rentable) el trabajar así. Aunque ciertos colectivos (como el mundillo de la Scene) vean mal el usar herramientas como Photoshop, lo cierto es que ahorran mucho trabajo y no hay necesidad de trabajar al modo antiguo (aunque al final tengamos que retocar pixel a pixel para un acabado perfecto). Esto sería comparable a que los programadores escribieran todo su código en Ensamblador; si bien es cierto que en algunas rutinas tengan que utilizarlo, pero en muchas otras no es rentable. Un ejemplo del buen hacer en animación de personajes lo tenemos en el videojuego de recreativas «Metal Slug» de Neo-Geo. El nivel de detalle alcanzado en todas sus animaciones justifica el grueso equipo de grafistas que hicieron el juego. Sin duda un buen ejemplo a seguir.

VEAMOS UN EJEMPLO...

Cuando comenzamos a dibujar, se nos plantea la duda de cual zona comenzar a dibujar, si nos centrarnos primero en la cara del personaje y luego seguir con el tronco y por último las extremidades, o si seguimos otro orden. Si empezamos a dibujar una parte, nos centramos en ella y, cuando hemos terminado y se ha dejado esta parte totalmente detallada, continuamos por otra parte cercana y seguimos el mismo proceso, corremos el riesgo de que el dibujo nos quede desproporcionado o torcido. Esto se debe a que, desde el principio, no hemos tenido una visión global del personaje que estábamos haciendo. Para evitar estos errores, avanzaremos en el dibujo por igual en todas sus partes. Es decir, no nos centraremos en ninguna parte en particular e iremos teniendo en todo momento una visión global del diseño.



LA CARA DEL LEON ESTA COLOREADA CON UN COLOR PLANO.

Supongamos que nos encargan dibujar un león vestido de Papá Noel (cosas más raras se han visto...). Hemos dividido el proceso de creación y coloreado del personaje en 5 fases. Las dos primeras fases (y la tercera en parte) las realizaremos en papel. En este primer ejemplo no animaremos nada todavía, por lo que no tendremos que usar la mesa de luz. Veremos los pasos a seguir para colorear un dibujo rápidamente sin tener que hacerlo pixel a pixel.

En la primera fase, realizaremos un esquema primario de nuestro personaje. Las articulaciones, por comodidad, las dibujamos con pequeñas esferas y los huesos serían los alambres que las unen. Este esquema nos valdrá para ver si la postura elegida da la sensación que queremos transmitir. Una vez que hemos elegido la postura adecuada, pasaremos a darle volumen a la figura en una segunda fase. Hay grafistas que prefieren hacer un esquema volumétrico en esta segunda fase (como si el modelo estuviera formado por conos, esferas y cubos; con su anatomía simplificada a esas figuras) pero nosotros nos hemos saltado este paso y hemos dibujado directamente el personaje con un nivel de detalle medio.

Cuando tengamos el dibujo a lápiz totalmente hecho (por cierto, para realizar los bocetos hemos utilizado lápices blandos; un 2B o HB vendría bien), limpiaremos la línea con bolígrafo negro de punta media (que no sea demasiado fina, un 0'4 vendría bien). Obtendremos en papel un dibujo como el de la fase 3, con las líneas totalmente definidas y sin correcciones. Escanearemos este dibujo y lo cargaremos en Photoshop.

Normalmente al escanear la imagen en pantalla aparecerán las líneas con tonos grises y con *pixel* de color gris claro que no tendrían que aparecer. Para limpiar y definir bien las líneas, aumentaremos un poquito el brillo (5%) y bastante más el contraste (50%) en la opción del menú principal *Image/Adjust/Brightness&Contrast*. Si la imagen actual no está en modo RGB, tendréis que ir a *Mode* y ponerla en RGB Color antes de empezar a colorear. Para que el coloreado se realice de la forma más rápida posible, cerraremos las superficies que queden abiertas con una línea del mismo color del que usaremos para a rellenar esa superficie y después, ayudados de la herramienta de relleno (*Paint Bucket*) con la opción de *anti-aliased* seleccionada, rellenaremos con su color correspondiente cada parte.

Para cerrar las superficies que vayan a ir del mismo color (como por ejemplo la cara del león), elegiremos el lápiz, con un grosor apropiado (no demasiado fino), activaremos la opción *darken* (con esto sólo pintamos del color seleccionado los *pixel* que son más claros que el color actual; así no pintaremos encima de las líneas negras que delimitan el dibujo) y elegiremos el color de la cara del león. Con el lápiz iremos cerrando la superficie desde dentro definida por la cara del león (figura 1), y asegurando que los ojos quedan fuera de la superficie, y que, al rellenar, no se saldrá el color fuera de la misma (figura 2). Si la superficie no es muy

grande, quizás tardemos menos coloreándola directamente con el lápiz (siempre con la opción *darken* seleccionada). Si al rellenar una superficie se quedan *pixel* sin colorear, podremos pasar por encima el pincel o el lápiz para colorearlos o quizás aumentando la tolerancia de la herramienta de relleno solucionemos el problema.
En este punto, cuando tengamos coloreado entero el personaje mediante colores planos, normalmente nos pararemos. Con un poco de práctica, estas cuatro primeras fases no nos llevarán más de diez minutos (y si disponemos de tabla digitalizadora, tan sólo cinco minutos). Cuando animemos no llegaremos a la fase de acabado con brillos y sombras porque supone un gran gasto de tiempo el animar correctamente con los brillos, ya que deberíamos seguirlos al milímetro para que no diera la impresión de que las luces cambian de lugar, ni se notaran efectos no deseados. Si acaso, llevaríamos alguna sombra en el modelo pero por lo general no estará tan acabado como en la fase 5. De cualquier forma, esto último que hemos dicho no tiene que tomarse al pie de la letra; si el juego para el que vamos a animar tiene un protagonista que aparece durante todo el juego en pantalla, deberá estar muy acabado, y podríamos gastar un poco más de tiempo en él que en el resto de personajes del juego (como el *Earth Worm Jim*, que cuenta con fantásticas animaciones del gusano que controlamos). En este ejemplo sí hemos terminado la imagen completamente. Para dar los brillos y las sombras necesarios hemos utilizado la herramienta de tono (la opción *burn* para las sombras y la opción *dodge* para los brillos). En ambos casos las hemos utilizado con tonos medios (*midtones*) excepto las zonas del traje que tenían partes blancas (como la bola del gorro y la parte inferior de los pantalones) donde hemos utilizado la opción de quemar con tonos altos (*highlights*). En esta última fase, hemos tardado otros 15 minutos más o menos.
Cuando estemos haciendo animaciones, trabajaremos con cada *frame* por separado del mismo modo que hemos hecho en este ejemplo (los coloreamos en el Photoshop porque la opción *darken* nos facilita mucho la labor), los reduciremos al tamaño apropiado para el juego, lo cargaremos en el Autodesk Animator Pro y retocaremos la imagen (ajustando la situación de cada *frame*, y retocando los *pixels* que no hayan quedado del todo bien). En el próximo capítulo del curso haremos nuestra primera animación en el Animator.

El encargo esta terminado, y en poco menos de media hora.

Y CON ESTO Y UN BIZCOCHO...

El próximo mes veremos un poco de teoría sobre anatomía y creación de personajes (proporciones, personalidad...) y haremos nuestra primera animación. Cuando se publique este artículo estaremos en enero de 1999. Si no sabéis qué pedirle a los Reyes Magos, desde aquí os damos un consejo; que os traigan una tabla digitalizadora porque ahorran mucho tiempo y hace que el trabajo sea más preciso. Sólo nos queda desear a todos feliz año nuevo y despedirnos... ¡Hasta el mes que viene!.

Y PARA VER...

Cuando vimos Toy Story nos quedamos impresionados por el buen trabajo que habían hecho los chicos de Pixar. Hace poco, Dream Works nos han demostrado que no todo estaba dicho en el mundo de la animación por ordenador. Sí, estamos hablando de *Antz*; posiblemente la mejor película de animación del 98.
Desde el comienzo la película te deja hipnotizado por sus logradas texturas. Al poco de haber empezado, Z (la hormiga protagonista) sirve de enganche para una bola formada por hormiguitas. En ese punto aparece el primer sistema de partículas (de hormigas) que obedece perfectamente a las leyes físicas cuando esta bola cae, rebota en el suelo, rueda y destroza algunas partes de la gruta... El animar a la perfección conjuntos de miles (o millones) de hormigas, cada una de las cuales da la impresión de moverse independientemente de las demás, fue un reto que superaron con creces los animadores de Dream Works. Otro de los puntos fuertes son las animaciones faciales, donde se utilizó un novedoso sistema basado en músculos y huesos simulando el humano, que no produce simplemente «caricaturas» de los movimientos que hacemos al hablar, sino que responde exactamente a los mismos.
En general la película deja muy buen sabor de boca ya que, además de lo comentado antes, la animación de los personajes está acompañada de multitud de detalles (como las animaciones del agua; simplemente geniales) que harán las delicias de todo aficionado a la animación por ordenador. El listón para próximas producciones lo han puesto alto. Para saber más acerca de *Antz*, puedes navegar por la página oficial de la película en http://www.antz.com, con información sobre el desarrollo de la misma. Es de visita obligatoria.